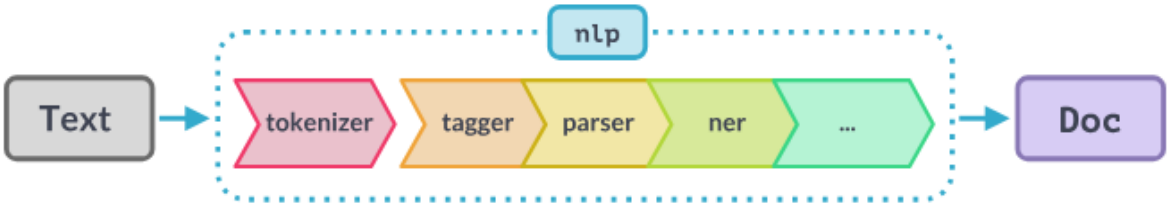
Natural Language Processing

Lab 4: Named Entity Recognition and Rule-Based Matching

Named Entity Recognition (NER)

Named-entity recognition is the task of locating and classifying named entities in text into predefined categories. A named entity is a "real-world object" that has a proper name (e.g. a person, organization, country, product, etc.). For example, in the sentence "Jim bought 300 shares of Acme Corp. in 2006." a NER system should identify "Jim" as a **Person**, "Acme Corp." as an **Organization**, and "2006" as a **Time**. Common entity types include **PERSON**, **ORG** (organizations), **GPE** (geopolitical entities like countries/cities), **LOC** (other locations), **PRODUCT**, **MONEY**, **DATE**, etc. SpaCy's pretrained English model (e.g. `en_core_web_sm`) recognizes many of these types; for example, it may label "Apple" as `ORG` and "U.K." as `GPE`. In spaCy, after processing text with a loaded model, the document's `doc.ents` attribute contains all identified entities as spans with text and label.

spaCy's NER



spaCy provides fast, pretrained statistical NER models for many languages. After loading a pipeline (e.g. `nlp = spacy.load("en_core_web_sm")`), calling `nlp(text)` produces a `Doc` whose `.ents` attribute lists recognized entities and labels. You can also use `spacy.explain(label)` to get a human-readable description of a label (e.g. `spacy.explain("ORG")` → "Companies, agencies, institutions, etc."). spaCy's NER is based on a deep learning model, so it predicts entities in context. Note that because models are statistical, they may make mistakes (false positives/negatives) if faced with unseen or ambiguous data.

After extracting entities, you can use spaCy's `displaCy` visualizer to see them in context. For example, `displacy.render(doc, style="ent")` produces an HTML/SVG visualization highlighting each entity and its label. (In a Jupyter notebook, you can display it inline with `jupyter=True`).

```
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")

text = ‘Nvidia is a GPU design company.’
doc = nlp(text)

for ent in doc.ents:
    print(ent.text, ent.label_)
displacy.render(doc, style='ent')
```

Task 1

Choose any paragraph (e.g., from a news website or Wikipedia) — at least 100 words. Print a frequency table of entity types (e.g., how many PERSON, ORG, GPE, etc.). Visualize entities using displaCy. Write a short reflection (3–5 sentences): Which entity types were most common? Were there any misclassifications?

Task 2

Using the paragraph from Task 1, filter all entities of type `ORG` and store them in a list. Write which entity is a misclassification.

Statistical vs. Rule-Based NER

Traditional NER can be approached with either statistical (machine learning) models or rule-based methods. **Statistical NER** (as used in spaCy’s built-in pipelines) relies on models trained on annotated corpora. These models learn patterns from data but require large labeled datasets and often treat NER as a sequence-labeling problem (e.g. using conditional random fields or neural networks). **Hand-crafted, rule-based** systems use lexicons or grammars (gazetteers, regular expressions, token patterns) and tend to achieve very high precision on the patterns they cover but at the cost of recall and development effort. In fact, hand-crafted grammar-based systems typically obtain better precision, but at the cost of lower recall and months of work by experienced linguists. Rule-based NER is practical when you have a finite list of entities (e.g. country names) or clear token patterns, and can be used to bootstrap a statistical model. It is often effective to combine approaches: use a statistical model for broad coverage and add rule-based patterns to catch special cases or domain-specific terms.

Rule-Based Matching in spaCy

In addition to statistical NER, spaCy offers powerful rule-based matching through the `Matcher` and `PhraseMatcher` classes. The `Matcher` lets you define patterns over token attributes (like `.text`, `.lower_`, part-of-speech tags, etc.) and find tokens/sequences that match these patterns (similar in spirit to regex on tokens). It can even trigger callbacks to modify the `Doc` (e.g. merge spans or assign labels). The `PhraseMatcher` is optimized for matching exact phrases: you provide a list of `Doc` or `Phrase` patterns, and it will quickly find those exact token sequences in text. In general, use `PhraseMatcher` when you have a known list of phrases to match (e.g. a list of product names), and use `Matcher` when you need more flexible, token-based rules (e.g. Noun + proper noun or number + capitalized word). Both can be used to implement rule-based NER or to find specific linguistic patterns. For example, you could use `Matcher` to find all sequences like `[{"LOWER":"hello"},{"IS_PUNCT":true},{"LOWER":"world"}]` in text. Because spaCy's matchers operate on the processed `Doc`, they give you access to token relationships and annotations (unlike raw regex).

```
from spacy.matcher import Matcher

matcher = Matcher(nlp.vocab)
pattern = [
    {"LOWER": "hello"},
    {"IS_PUNCT": True},
    {"LOWER": "world"}
]
matcher.add("HelloWorld", [pattern])

doc = nlp("Hello, world! Hello world. Not a match: hello world without
punctuation.")
matches = matcher(doc)
for match_id, start, end in matches:
    span = doc[start:end]
    match_name = nlp.vocab.strings[match_id]
    print(match_name, "→", span.text)
```

```
from spacy.matcher import PhraseMatcher

phrases = ["New York", "IBM", "Google"]
```

```
phrase_patterns = [nlp.make_doc(text) for text in phrases]
phrasematcher = PhraseMatcher(nlp.vocab, attr="LOWER")
phrasematcher.add("TechCompany", phrase_patterns)

doc = nlp("IBM released a new product in New York City yesterday.")
matches = phrasematcher(doc)
for match_id, start, end in matches:
    span = doc[start:end]
    print(doc.vocab.strings[match_id], "→", span.text)
```

Task 3

Take a paragraph from the internet that includes domain-specific terms (e.g., names of software tools, diseases, or local organizations). Use spaCy's NER to check which ones it fails to recognize. Use the Matcher or PhraseMatcher to create patterns for at least three such terms. Print the matches and confirm they are correctly identified. Explain why these terms might not appear in the pretrained model's entity vocabulary.

Custom Entity Labeling

spaCy also lets you add your own named entities. The built-in `EntityRuler` component provides a convenient way to assign labels via patterns. The `EntityRuler` takes dictionaries with **"pattern"** and **"label"** and, when applied in the pipeline, will add matches as new **Span** entities in `doc.ents`. For example, you can define `{"label": "LANGUAGE", "pattern": "Python"}` so that every occurrence of **Python** is tagged as a `LANGUAGE` entity. You can insert an `EntityRuler` before the statistical NER in the pipeline to ensure your rules take precedence. Alternatively, you can manually create a Span and append it to `doc.ents`.

```
from spacy.pipeline import EntityRuler

ruler = EntityRuler(nlp, overwrite_ents=True)
ruler.add_patterns([{"label": "LANGUAGE", "pattern": "Python"}])
nlp.add_pipe(ruler, before="ner")

doc = nlp("He writes code in Python and uses SpaCy for NER.")
for ent in doc.ents:
    print(ent.text, ent.label_)
```

```
from spacy.tokens import Span

doc = nlp("He learned Python for data science.")

span = Span(doc, 2, 3, label="LANGUAGE")
doc.ents = list(doc.ents) + [span]
print(doc.ents)
```

Task 4

Add an EntityRuler to your NLP pipeline (before the NER component). Create at least five custom patterns for entities in your field of interest (e.g., `{"label": "TECH_PRODUCT", "pattern": "iPhone"}`). Apply your pipeline on a sample paragraph containing these entities. Print all entities (both default and custom).

Task 5

Prepare a short text (around 150 words) that includes both standard and domain-specific entities. Run spaCy's default NER and record detected entities. Apply your custom Matcher or EntityRuler patterns on the same text.

Count:

- True Positives (entities correctly identified by both methods)
- False Negatives (entities missed by the model but caught by your rules)
- False Positives (incorrect entities caught by either)

Write a short note comparing performance: Where did the model fail? Where did your rules help?